Burrows-Wheeler transform acceleration based on CUDA

Chang Sheng, Fengzhi Dai*
Tianjin University of Science and Technology, Tianjin, China
*E-mail: * daifz@tust.edu.cn*
www.tust.edu.cn

Abstract

Burrows-Wheeler transform (BWT) is a commonly used transform in compression or text comparison. For example, in bzip2, BWT is used to preprocess the original data, then the same characters in the original data are close to each other, which improves the compression rate. Because the prefix tree of the original string can be easily obtained from the result of the BWT, BWT is also applied to the search and comparison of strings. For instance, the comparison of DNA sequences uses the BWT algorithm. However, BWT is not a fast algorithm, only tens of megabytes per second on CPU. This article uses the GPU to sort the original string by the base of the 4-byte key size radix sort. After radix sort, the part with insufficient length is sorted again to complete the BWT algorithm.

Keywords: BWT, acceleration, CUDA, GPU

1. Introduction

Burrows-Wheeler transform (BWT) is a data compression algorithm proposed by Burrows and Wheeler in 1994 [1]. It can be used before other compression algorithms, such as MTF, Huffman, and RLE, so that these zero-order entropy codes can achieve the effect of high-order entropy coding. Therefore, it can replace the LZ77 sliding window search algorithm before the general compression algorithm. Moreover, thanks to the restoration method of BWT transformation, we can obtain the prefix tree of the original string through the result of BWT transformation. Through the prefix tree, the search for the string may be completed faster, and the result of the BWT transformation takes up less space than using the prefix tree. Therefore, BWT transformation is used in bzip2 compression and DNA sequence alignment.

For the implementation of the BWT algorithm, first, add a terminator to the original string, so that the starting position can be found when restoring. For example, for the string "aabcg", add the terminating character '#' to get a string of length 6 "aabcg#", and then perform a circular shift to get 6 strings, then sort, and the last column of the sorted result is the transformed result "g#aabc", as shown in Table 1.

Table 1. BWT Algorithm

Rotate left	Sort	first column	last column
aabcg#	#aabcg	#	g
abcg#a	aabcg#	a	#
bcg#aa	abcg#a	a	a
cg#aab	bcg#aa	b	a
g#aabc	cg#aab	c	b
#aabcg	g#aabc	g	c

The last column is the result. The method of restoring the string is to start with the character '#' in the first column of Table 1, get the first 'g' in the last column, then find the first 'g' in the first column, and repeat this operation to get '#gcbaa', then reverse the arrangement to get the original string.

In the process of performing the BWT algorithm, we can find that the original algorithm needs to occupy

O(m*m) space. Of course, this can be turned into O(m) space through the position index. So choosing the suitable string sorting algorithm becomes the main point of acceleration. If a complete string comparison is performed each time, memory access will be discontinuous, so using 4 bytes as a key, and indexing the corresponding value, sorting the key-value pairs can increase the continuity of memory access. For the sorting algorithm, it is realized by selecting the radix sorting that is convenient to run on the GPU. Experiments show that this method is effective.

2. Main ideas

Fig.1 is the Implementation process.

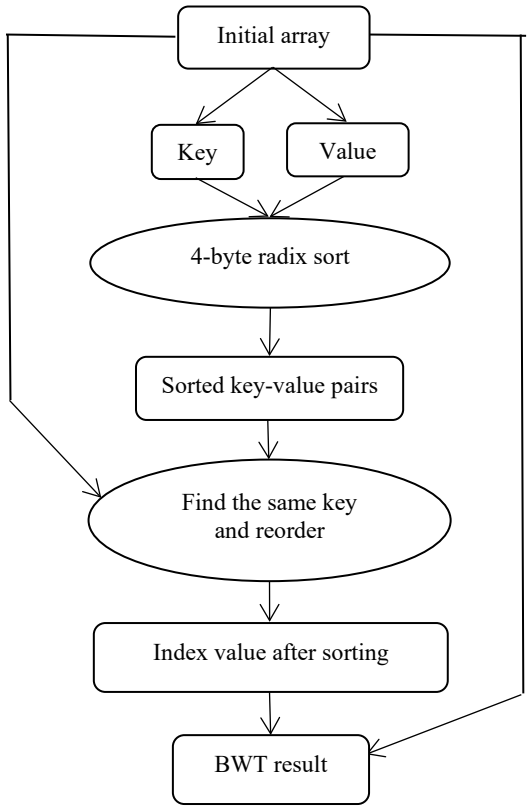

Fig.1 Implementation process

In order to complete the BWT algorithm, first, use 100M data to generate 4-byte keys and values, Then use the radix sort to sort the generated key-value pairs according to the key. For the case where the key values are the same because of the limited length, re-sort them, and finally use the value corresponding to the sorted key to generate the BWT result.

3. Implementation details

For the above process, the specific implementation can be divided into the following 4 parts.

3.1. Generate key-value pairs

This saves space and is a step towards shifting to key-value pair sorting. The key is the first 4 bytes of m*m strings, and the value is the number of rows corresponding to m*m strings. So 3 bytes from the current position, plus the current byte, a total of 4 bytes are keys, and the data from 0 to 100000000-1 is the value.

It is worth noting that the memory access must be performed in a continuous manner as shown in Fig.2, otherwise the performance will be greatly reduced. The subsequent algorithms in this article use this method when this access is optional.

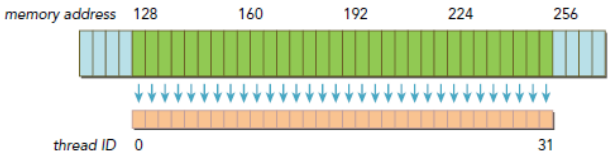

Fig.2 Memory access method

3.2. Key-value pairs radix sort

By sorting the key-value pairs, most of the data can be in the correct order. For the sorting algorithm, radix sorting is selected here. Because the radix sort has a time complexity of O(nlogn) when the radix size is determined, and occupies 2n space, the algorithm is simple and regular, suitable for implementation by GPU, and has been implemented on GPU.

3.3. Same key recognition and reorder

Because the 4-byte fixed-length mode is used, and the actual string length is 100M, it needs to be compared again when the 4-byte keys are equal. This program compares all keys. If they are not equal, the value in the key-value pair is used to index the original data, and the comparison is performed until the result is different. This

algorithm sorts multiple data with smaller length at the same time.

Because the comparison of indefinite length is performed, the comparison sorting algorithm is used for the reordering part. Merge sort is considered for parallel sorting of the remaining data. However, because the merge sort has low parallelism in the final stage, and some data has a large length, the bitonic sort algorithm is used for the part with a length greater than 2000. It has high parallelism and is suitable for sorting small-length data.

3.4. Generate BWT results

After the previous step, the sorted keys have been obtained. According to the corresponding value, the sorted result of the original m*m strings can be obtained. By formula (1), the result corresponding to the last column of Table 1 can be obtained.

$$pos2 = (pos1 + L - 1)\%L \tag{1}$$

In formula (1), pos2 is the result of the last column, pos1 is the index value obtained after sorting, L is the length of the original data, % is the same as that represented in the C language for taking the remainder.

In this way, the BWT results can be obtained. Because the data is sorted, the access to the original string becomes random access, so this step consumes time.

4. Experimental results

In order to determine the difference in performance of various graphics cards, the sensitivity to data and the internal performance of the algorithm, the following experiment was carried out.

4.1. Performance on various graphics cards

In order to obtain the performance of this algorithm on different graphics cards and data volumes, the following tests were performed with random data. The result is shown in Fig.3.

It can be seen that when the amount of data is small, the concurrency of the GPU is not enough and the speed is reduced. As the amount of data increases, the speed begins to rise, and for random data, as the amount of data further rises, the speed does not decrease.

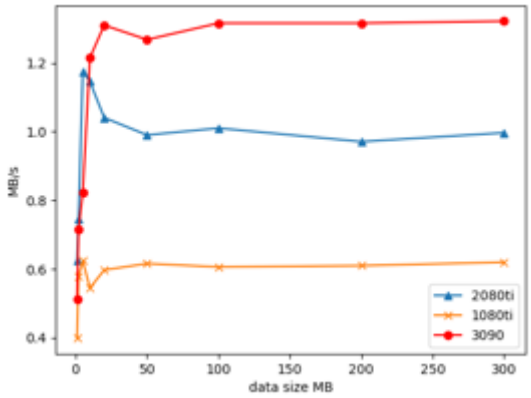

Fig.3 Performance on different graphics cards

4.2. Sensitivity to data

In order to obtain the running performance on different data, tests were carried out. The test data includes random data, win7 installation iso format files, molecular dynamics simulation trajectory trr format files, and human genome fasta format files. With 100MByte data volume, the data of Table 2 was obtained on 2080ti.

Table 2. Running performance

File type	Speed
Random data	**1.01GB/s**
Win7 iso	**0.998GB/s**
Trajectory trr	**0.549GB/s**
Genome fasta	**0.202GB/s**

For genetic data, because there are only 4 kinds of characters per byte, 16 characters are used to generate a 4-byte key, which has an impact on speed.

When there is a certain repetition in the data, this algorithm has appropriate sensitivity to the data. It will increase the amount of reordered data and cause too many character comparisons during comparison. Using Manber-Myers multiplication algorithm [2] can reduce data sensitivity, but will increase the amount of calculation for less repeated data.

4.3. Algorithm internal performance

In order to understand the time consumption of each part of the algorithm, a test was performed on 2080ti using random data with a data volume of 100MByte. The results are shown in Table 3.

Table 3. Time consumption

Stage	Time	Speed
Generate key-value pairs	1.9ms	52.6GB/s
Key-value pairs radix sort	62ms	1.61GB/s
Recognition and reorder	5.67ms	17.6GB/s
Generate BWT results	29ms	3.44GB/s

Time is mainly used to sort the key-value pairs, which is an important step in ordering the data. It also takes a certain amount of time to generate the result, because the access to the original string becomes random after the sort. In the reordering phase, because the amount of reordered data on random data is less, and the length of the reordering comparison is similar, it does not take much time.

5. Conclusion

The algorithm sorts 4-byte key-value pairs by radix sort. It can sort most of the data with less repetitiveness, and then use the reordering method to sort the remaining data. Great performance improvement for data with low repeatability. For data with high repeatability and uneven distribution, the speed is higher than the CPU. Using the multiplication algorithm can reduce data sensitivity [3], and this algorithm has more advantages for data with low repeatability.

References

1. Michael Burrows, David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994
2. Udi Manber, Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. Siam Journal on Computing, 1993, 22(5): pp.935-948.
3. Kartsev, Petr F. High performance OpenCL realization of Burrows-Wheeler transform on GPU. International Workshop on OpenCL, Oxford, ENGLAND, May. 2018, pp. 83-84.